ASYNCHRONE ABARBEITUNG VON TASKS ÜBER CHANNELS

Aufgabenverteilung

Channel<T>, damit voneinander abhängige Dienste autark arbeiten können.

Sollen innerhalb eines Prozesses verschiedene voneinander abhängige Dienste weitestgehend autark arbeiten können, bietet es sich an, die Zwischenergebnisse über Queues (First-in-First-out-Listen) von einem zum nächsten Dienst zu übertragen. Seit .NET Core 3.0 gibt es neben den klassischen Auflistungsklassen wie *Queue* oder *ConcurrentQueue* den bislang noch recht unbekannten Typ *Channel<T>*, der speziell für diese Aufgabenstellung entwickelt wurde.

In vielen technischen Prozessen muss man auf asynchron eingehende Informationen reagieren. Das können beispielsweise Messwerte sein, die verarbeitet und angezeigt werden sollen, oder Bestellungen, die abzuarbeiten sind. Verschiedene Arbeitsschritte sind zu durchlaufen, können aber oftmals parallel abgearbeitet werden. So muss die Verarbeitung einer Bestellung (Zusammensuchen der Ware und Verpacken derselben) nicht mit der nächsten Bearbeitung warten, bis der Logistiker das Paket ausgeliefert hat, sondern kann unabhängig vom Versand weitere Aufträge abarbeiten.

Lagert man die unterschiedlichen Arbeitsschritte in Tasks aus, dann sind die Übergabepunkte der Zwischenergebnisse typische Schwachstellen der asynchronen Ausführung. Zugriffe auf Listen, in denen ein- und ausgehende Daten gespeichert werden, müssen Thread-sicher implementiert werden. Das in eigener Regie über Locks oder Ähnliches zu lösen, führt schnell zu unübersichtlichen und unhandlichen Ansätzen, die bei komplexeren Systemen oft dazu führen, dass sich Tasks gegenseitig blockieren. Schlimmer noch – die verwendeten Lock-Mechanismen blockieren meist die zugrunde liegenden Threads, und wenn diese wie bei *Task.Run()* aus dem ThreadPool entnommen werden, kommt schnell alles zum Erliegen.

First-in-First-out-Listen (FiFo-Listen), auch Queues genannt, bieten sich hier als komfortable Lösung an. Der Produzent speichert seine Ergebnisse in der Queue und ein anderer Prozess kann diese dann später aus der Queue entnehmen und weiterverarbeiten. Nehmen wir als Beispiel ein paar Datenklassen für einen fiktiven Versandhandel (Listing 1). Eine Reihe von Artikeln soll gemäß eingehender Bestellungen an die Kunden versendet werden. Die Bestellungen kommen asynchron an, hier simuliert durch eine *IAsyncEnumerable*-Implementierung (Listing 2). Die Implementierung als *IAsyncEnumerable* wurde hier nur zu Demonstrationszwecken verwendet. Alternativ könnten die Daten auch über Events, Callbacks und so weiter in der Anwendung auflaufen.

In Listing 3 sehen Sie einen klassischen Lösungsansatz unter Verwendung von *ConcurrentQueue*-Instanzen. *ConcurrentQueue* bildet eine FiFo-Liste ab und beinhaltet, wie der Name schon vermuten lässt, die Synchronisation der Zugriffe für Push (*Enqueue*) und Pull (*Dequeue*), sodass man sich bei der Programmierung hier nicht selbst um die Thread-Sicherheit kümmern muss. Das Hinzufügen neuer Daten über *Enqueue* ist für den Produzenten völlig unproblematisch. Er

Listing 1: Datenklasse für die Beispiele

```
public record Artikel(string Bezeichnung,
  int Verpackungszeit);
public record Adresse(string Ort, int Versandzeit);
public record Bestellung(int Auftragsnummer,
  IEnumerable<Artikel> Artikel, Adresse Adresse);

public class Auftragseingang {
  private readonly Artikel[] artikelliste =
      new Artikel[] {
    new Artikel("Hemd",2),
    new Artikel("Geschirr",5),
    new Artikel("Bild",3),
    new Artikel("Ball",1),
    new Artikel("Fernseher",4),
    new Artikel("Spiegel",4)
  };
  private readonly Adresse[] adressen =
      new Adresse[] {
    new Adresse("Köln",1),
    new Adresse("München",1),
    new Adresse("Hamburg",1),
    new Adresse("Bremen",1),
    new Adresse("Dresden",1),
    new Adresse("Wien",2),
    new Adresse("Brüssel",2),
    new Adresse("London",3),
    new Adresse("Wellington",5),
    new Adresse("Tokyo",3),
    new Adresse("Ushuaia",5),
    new Adresse("Stanley",8),
    new Adresse("Edinburgh of the Seven Seas",30)
  };
```

Listing 2: Generator für Bestellungseingänge

```
/// <summary>
/// Bestellungen generieren und als AsyncEnumerable
/// bereitstellen
/// </summary>
/// <param name="anzahl">Anzahl der gewünschten
/// Bestellungen</param>
/// <param name="cancellationToken">Abbruch-Token
/// </param>
/// <returns>Bestellungen</returns>
public async IAsyncEnumerable<Bestellung>
    GetBestellungen(int anzahl,
    [EnumeratorCancellation] CancellationToken
    cancellationToken = default)
{
  int auftragsnummer = 0;

  // Gewünschte Anzahl Bestellungen generieren
  while (anzahl >= 1)
  {
    anzahl--;
    auftragsnummer++;

    // try/catch nur, damit der Debugger
    // hier nicht stoppt
    try
    {
      // Abstand zwischen den Bestelleingängen
      // simulieren
      await Task.Delay(300, cancellationToken);
    }
    catch (Exception)
    {
      Console.WriteLine(
        "Bestellungsannahme abgebrochen");
      yield break;
    }

    // Artikel zusammenstellen
    var n = Random.Shared.Next(1, 10);
    var artikel = new Artikel[n];
    for (int i = 0; i < n; i++) artikel[i] =
      artikelliste[Random.Shared.Next(
      artikelliste.Length)];

    // Bestellung vervollständigen
    var bestellung = new Bestellung(
      auftragsnummer, artikel,
      adressen[Random.Shared.Next(adressen.Length)]);

    // aktuelle Iteration abschließen
    yield return bestellung;
  }
}
```

```
00.005 Bearbeitung der Bestellungen begonne
00.007 Bestellungen entgegennehmen
00.326 Auftrag 1 in queue verschoben
00.326 Beginn Auftrag 1 verpacken
00.636 Auftrag 2 in queue verschoben
00.948 Auftrag 3 in queue verschoben
01.261 Auftrag 4 in queue verschoben
01.562 Auftrag 5 in queue verschoben
01.870 Auftrag 6 in queue verschoben
02.172 Auftrag 7 in queue verschoben
02.434 Ende Auftrag 1 verpacken
02.435 Beginn Auftrag 2 verpacken
02.482 Auftrag 8 in queue verschoben
02.792 Auftrag 9 in queue verschoben
03.105 Auftrag 10 in queue verschoben
03.105 Alle Bestellungen entgegengenommen
04.135 Ende Auftrag 2 verpacken
04.136 Beginn Auftrag 3 verpacken
06.751 Ende Auftrag 3 verpacken
06.752 Beginn Auftrag 4 verpacken
09.058 Ende Auftrag 4 verpacken
09.058 Beginn Auftrag 5 verpacken
10.560 Ende Auftrag 5 verpacken
10.561 Beginn Auftrag 6 verpacken
11.763 Ende Auftrag 6 verpacken
11.763 Beginn Auftrag 7 verpacken
15.265 Ende Auftrag 7 verpacken
15.265 Beginn Auftrag 8 verpacken
17.167 Ende Auftrag 8 verpacken
17.167 Beginn Auftrag 9 verpacken
18.881 Ende Auftrag 9 verpacken
18.881 Beginn Auftrag 10 verpacken
21.483 Ende Auftrag 10 verpacken
```

Klassischer Queue-Ansatz – hier nur der Verpackungs-Schritt (Bild 1)

wird hier nicht spürbar ausgebremst. Schwieriger ist die Entnahme der Daten über *Dequeue*, denn hier müssen zwei Fälle unterschieden werden:

Entweder sind Daten vorhanden, dann heißt es diese zu nehmen und zu verarbeiten, oder es liegen keine Daten vor, dann ist nichts zu tun. Der zweite Punkt wirft jedoch ein Problem auf. Wenn nichts zu tun ist, wann soll denn dann die nächste Abfrage erfolgen?

Würde man in einer *while*-Schleife kontinuierlich die Queue fragen, ob es was Neues gibt, würde sich daraus eine unnötige CPU-Last auf dem ausführenden Thread ergeben. Fügt man Verzögerungen zwischen den Abfragen ein, kann vielleicht nicht schnell genug auf neue Daten reagiert werden. Benötigt wird also ein Steuermechanismus, der dazu führt, dass auf der Empfängerseite der Queue der Code erst dann ausgeführt wird, wenn auf der Senderseite etwas hinzugefügt worden ist. Man braucht also ein weiteres Synchronisationsobjekt.

Im Beispiel wird hierzu ein Semaphor verwendet. Dazu gibt es im .NET-Framework eine Lightweight-Implementierung mit dem Namen *SemaphoreSlim*. Anders als bei den meisten anderen Synchronisationsobjekten verfügt *SemaphoreSlim* neben den üblichen blockierenden *Wait*-Aufrufen auch über einen Task-basierten Mechanismus, der hier wesentlich ▶

besser passt. *SemaphoreSlim.WaitAsync* gibt ein *Awaitable* zurück, das für den asynchronen Aufruf in

```
await semaphoreSlim.WaitAsync();
```

verwendet werden kann. So wird nur das Task-Objekt bis zum Eintreffen neuer Daten suspendiert, nicht aber der ausführende (ThreadPool-)Thread blockiert. Der Produzent signalisiert die Verfügbarkeit neuer Daten über

Listing 3: Klassischer Ansatz

```
// Ansatz mit ConcurrentQueue und Semaphor
// Beispieldatengenerator
static readonly Auftragseingang
  auftragseingang = new();
private static void BeispielMitConcurrentQueue()
{
  ConcurrentQueue<Bestellung> queue = new();
  SemaphoreSlim semaphoreSlim = new(0);
  Task.Run(async () =>
  {
    Print("Bestellungen entgegennehmen",
      ConsoleColor.Yellow);
    await foreach (var bestellung in
        auftragseingang.GetBestellungen(10))
    {
      Print($"Auftrag {bestellung.Auftragsnummer} in
        queue verschoben", ConsoleColor.Yellow);
      queue.Enqueue(bestellung);
      semaphoreSlim.Release();
    }
    Print("Alle Bestellungen entgegengenommen",
      ConsoleColor.Yellow);
  });
  Task.Run(async() =>
  {
    Print("Bearbeitung der Bestellungen begonnen",
      ConsoleColor.Blue);
    while (true) {
      await semaphoreSlim.WaitAsync();
      if(queue.TryDequeue(out var bestellung)) {
        Print($"Beginn Auftrag {
          bestellung.Auftragsnummer} verpacken",
          ConsoleColor.Blue);
        await Task.Delay(bestellung.Artikel.Sum(
          a => a.Verpackungszeit) * 100,
          cancellationToken);
        Print($"Ende Auftrag {
          bestellung.Auftragsnummer} verpacken",
          ConsoleColor.Blue);
      }
    }
  });
  Console.ReadLine();
}
```

Listing 4: ChannelWriter

```
// Ein ChannelWriter abstrahiert das Speichern in
// der Queue
// Channel für die Auftragsbearbeitung (Verpacken)
static readonly Channel<Bestellung>
  channelAuftragseingang =
  Channel.CreateUnbounded<Bestellung>();
// Token (-generator) für Abbruch durch den Benutzer
static CancellationTokenSource
  cancellationTokenSource = new();
static CancellationToken cancellationToken =
  cancellationTokenSource.Token;

static async Task BestellungenVerarbeiten()
{
  Print("Bestellungen entgegennehmen",
    ConsoleColor.Yellow);
  await foreach (var bestellung in
    auftragseingang.GetBestellungen(5,
    cancellationToken))
  {
    Print($"Bestellung {bestellung.Auftragsnummer} mit
      {bestellung.Artikel.Count()} Artikeln (
      {string.Join( ",",bestellung.Artikel.Select(a=>
      a.Bezeichnung))}) nach
      {bestellung.Adresse.Ort}", ConsoleColor.Yellow);
    try
    {
      await channelAuftragseingang.Writer.WriteAsync(
        bestellung, cancellationToken); }
    catch (TaskCanceledException) {
      Print("BestellungenVerarbeiten abgebrochen",
        ConsoleColor.Red);
    }
  }
  channelAuftragseingang.Writer.Complete();
  Print("Alle Bestellungen entgegengenommen",
    ConsoleColor.Yellow);
}
```

```
semaphoreSlim.Release();
```

Bild 1 zeigt beispielhaft die Ausgabe einer Abarbeitungsfolge nach diesem Muster. Auch wenn der Ansatz prinzipiell funktioniert, muss doch vieles von Hand organisiert werden. Auf der Empfängerseite wäre vielleicht statt der Endlosschleife eine asynchrone *foreach*-Konstruktion schöner, und auch Abbruchmöglichkeiten über *Cancellation*-Tokens müssten noch ergänzt werden.

Außerdem wäre es ganz praktisch, die Implementierungen für Produzenten und Konsumenten als Writer und Reader sauberer trennen zu können.

Channel<T>

Hier kommt jetzt die neue *Channel*-Klasse aus dem Namensraum *System.Threading.Channels* ins Spiel. Sie kommt bereits mit einer fertigen Implementierung von *ConcurrentQueue*, Semaphor-Synchronisation, Reader-/Writer-Objekten, *CancellationToken*-Berücksichtigung sowie verschiedenen Optimierungsmöglichkeiten für unterschiedliche Einsatzfälle daher.

Eingeführt wurden die Channels bereits in .NET Core 3.0. Aufgrund der zahlreichen Neuerungen, die in .NET und C# seitdem hinzugekommen sind, sind sie aber vielen Entwicklern verborgen geblieben. Dokumentationen zu den *Channel*-Klassen finden sich in [1] und [2], ein paar aufschlussreiche Videos unter [3] und [4].

Channel<T> ist ein abstrakter Typ, den Sie selbst nicht instanzieren können. Stattdessen legen Sie eine *Channel*-Instanz mithilfe einer statischen Methode der nicht abstrakten Basisklasse *Channel* an. Hier gibt es grundsätzlich zwei Varianten zu unterscheiden: ▶

Listing 5: ChannelReader

```
// Ein ChannelReader abstrahiert verschiedene Ansätze
// für Lesevorgänge (hier als IAsyncEnumerable) aus
// der Queue

// Channel für den Versand
static readonly Channel<Bestellung>
  channelVersandInland =
  Channel.CreateBounded<Bestellung>(
  new BoundedChannelOptions(5) {
  SingleReader = true });

static readonly Channel<Bestellung>
  channelVersandAusland =
  Channel.CreateBounded<Bestellung>(2);
...

static async Task VerpackungsteamStarten(string name)
{
  Print($"Verpackungsteam {name} hat Arbeit begonnen",
    ConsoleColor.Green);

  try
  {
    await foreach (var bestellung in
        channelAuftragseingang.Reader.ReadAllAsync(
        cancellationToken))
    {
      await Verpacken(name, bestellung);
    }

    Print($"Verpackungsteam {name} ist fertig",
      ConsoleColor.Green);
  }
  catch (OperationCanceledException)
  {
    Print($"Verpackungsteam {name} hat die Arbeit
      abgebrochen", ConsoleColor.Red);
  }
}

static async Task Verpacken(string teamname,
    Bestellung bestellung)
{
  Print($"Verpackung von Auftrag
    {bestellung.Auftragsnummer} durch {teamname}
    begonnen", ConsoleColor.Magenta);
  await Task.Delay(bestellung.Artikel.Sum(
    a => a.Verpackungszeit) * 100, cancellationToken);
  Print($"Verpackung von Auftrag
    {bestellung.Auftragsnummer} durch {teamname}
    beendet, jetzt versenden", ConsoleColor.Magenta);

  if (bestellung.Adresse.Versandzeit > 1)
  {
    await
      channelVersandAusland.Writer.WriteAsync(
      bestellung, cancellationToken);
    Print($"Auftrag {bestellung.Auftragsnummer} ins
      Ausland versendet", ConsoleColor.Blue);
  }
  else
  {
    await
      channelVersandInland.Writer.WriteAsync(
      bestellung, cancellationToken);
    Print($"Auftrag {bestellung.Auftragsnummer} ins
      Inland versendet", ConsoleColor.Blue);
  }
}
```

- *CreateBounded* generiert einen BoundedChannel,
- *CreateUnbounded* generiert einen UnboundedChannel.

Ein UnboundedChannel entspricht in etwa unserem ersten Ansatz. Die Anzahl der speicherbaren Datenpakete ist unbegrenzt. Der Produzent (Writer) kann, ohne blockiert zu werden, beliebig viele Daten in die Queue hineinschieben. Als Nachteil könnte sich hierbei ergeben, dass die Zahl der Einträge sehr groß wird, wenn der beziehungsweise die Reader nicht schnell genug mit der Verarbeitung nachkommen. Um das zu verhindern, kann man auf einen BoundedChannel zurückgreifen. Bei diesem gibt man die maximale Anzahl von Einträgen fix vor. Dann allerdings muss der Writer blockiert werden, wenn er neue Einträge speichern möchte und die Queue bereits voll ist. Oder der Channel wird so eingestellt, dass Daten verworfen werden. Dazu später mehr. Es bleibt daher je nach Einsatzfall abzuwägen, welches die bessere Variante ist.

Gib mir alles

Über den *Typ*-Parameter der beiden *Create*-Methoden wird der zu speichernde Datentyp festgelegt. Dies ist dann auch der Typ des generierten *Channel<T>*-Objekts. Im Beispiel in Listing 4 kommt zunächst ein UnboundedChannel zum Einsatz. Beim Aufruf von *CreateUnbounded* können Vorgaben für die interne Optimierung des Channels übergeben werden (Parameter vom Typ *UnboundedChannelOptions*). Wenn sichergestellt ist, dass nur ein einziger Writer Daten einträgt, müssen die Schreibzugriffe nicht synchronisiert werden. Das Setzen von *SingleWriter = true* unterdrückt diese Synchronisation und spart Ausführungszeit. Wenn es dann allerdings doch konkurrierende Schreibzugriffe gibt, wird das früher oder später zu Problemen führen.

Analog dazu kann man festlegen, dass es keine konkurrierenden Lesevorgänge geben wird (*SingleReader=true*). Auch das muss man im Programm selbst gewährleisten. Ferner lässt sich noch über *AllowSynchronousContinuations* steuern, ob nach den *async-await*-Aufrufen im selben Thread fortzufahren ist (Standardeinstellung, wichtig zum Beispiel bei Zugriffen im UI-Thread) oder nicht (bewirkt eine bessere Performance).

Über die Eigenschaft *Writer* des Channels können Daten in die Queue hineingeschoben werden. Die einfachste Variante für den Schreibvorgang ist in diesem Fall ein Aufruf wie

```
await channel.WriteAsync(...)
```

Eine längerfristige Blockade des Tasks ist hier trotz *await* nicht zu erwarten, da bei einem *UnboundedChannel* maximal konkurrierende Schreibzugriffe synchronisiert werden müssen. Ansonsten sind alle Schreibzugriffe jederzeit erlaubt.

Alternativ kann man via *TryWrite* versuchen, Daten in die Queue einzutragen, muss im Fehlerfall (Rückgabewert *false*) sich dann aber selbst um die Wiederholung des Vorgangs kümmern.

In vielen Situation wird ein Channel nicht endlos lang geöffnet bleiben. Irgendwann ist die Arbeit erledigt und alle Aufträge abgearbeitet. Wenn der Produzent dann keine neuen Daten mehr bereitstellen wird, ist es sinnvoll, den Channel zu schließen. Das lässt sich über einen Aufruf der Methode *Complete* beziehungsweise *TryComplete* bewerkstelligen.

Zum Lesen aus dem Channel wird ein *ChannelReader*-Objekt benötigt, das der Channel über die Eigenschaft *Reader* bereitstellt. Auch hier gibt es wieder verschiedene Möglichkeiten, den Reader zu benutzen. Im Beispiel in Listing 5 wird über *Reader.ReadAllAsync* ein *IAsyncEnumerable* abgeru-

Listing 6: BoundedChannels und UnboundedChannels

```
// Der Zugriff auf BoundedChannels oder
// UnboundedChannels ist weitestgehend identisch
private static async Task PaketversandAusführen(
    string name, Channel<Bestellung> channelVersand)
{
  try
  {
    Print($"Logistiker ({name}) bereit zum Versenden
      der Pakete", ConsoleColor.Cyan);
    await foreach (
        var item in channelVersand.Reader.ReadAllAsync(
        cancellationToken))
    {
      Print($"Versand für Auftrag
        {item.Auftragsnummer} nach
        {item.Adresse.Ort} begonnen. Dauer [Tage]:
        {item.Adresse.Versandzeit}",
        ConsoleColor.Cyan);
      await Task.Delay(
        item.Adresse.Versandzeit * 1000,
        cancellationToken);
      Print($"Ware von Auftrag {item.Auftragsnummer}
        in {item.Adresse.Ort} ausgeliefert",
        ConsoleColor.Cyan);
    }
    Print($"Logistiker ({name}) hat alle Versand-
      aufträge abgeschlossen",
      ConsoleColor.Cyan);
  }
  catch (OperationCanceledException)
  {
    Print("Logistiker ({name}) hat die Arbeit
      abgebrochen",
      ConsoleColor.Red);
  }
}
```

```
Zum Beenden Eingabetaste drücken
00.007 Bestellungen entgegennehmen
00.012 Verpackungsteam A hat Arbeit begonnen
00.014 Verpackungsteam B hat Arbeit begonnen
00.015 Verpackungsteam C hat Arbeit begonnen
00.016 Logistiker (Inland) bereit zum Versenden der Pakete
00.016 Logistiker (Ausland) bereit zum Versenden der Pakete
00.319 Bestellung 1 mit 8 Artikeln (Spiegel,Bild,Spiegel,Hemd,Fernseher,Fernseher,Geschirr,Ball) nach Wien
00.339 Verpackung von Auftrag 1 durch A begonnen
00.627 Bestellung 2 mit 2 Artikeln (Fernseher,Ball) nach Dresden
00.628 Verpackung von Auftrag 2 durch C begonnen
00.941 Bestellung 3 mit 9 Artikeln (Bild,Ball,Hemd,Spiegel,Hemd,Hemd,Geschirr,Fernseher,Spiegel) nach Wien
00.941 Verpackung von Auftrag 3 durch B begonnen
01.129 Verpackung von Auftrag 2 durch C beendet, jetzt versenden
01.129 Auftrag 2 ins Inland versendet
01.130 Versand für Auftrag 2 nach Dresden begonnen. Dauer [Tage]: 1
01.252 Bestellung 4 mit 9 Artikeln (Fernseher,Ball,Hemd,Spiegel,Spiegel,Spiegel,Hemd,Geschirr,Geschirr) nach München
01.253 Verpackung von Auftrag 4 durch C begonnen
01.566 Bestellung 5 mit 9 Artikeln (Geschirr,Fernseher,Hemd,Geschirr,Ball,Geschirr,Spiegel,Bild,Ball) nach Edinburgh of the Seven Seas
01.567 Alle Bestellungen entgegengenommen
02.142 Ware von Auftrag 2 in Dresden ausgeliefert
03.052 Verpackung von Auftrag 1 durch A beendet, jetzt versenden
03.053 Auftrag 1 ins Ausland versendet
03.053 Verpackung von Auftrag 5 durch A begonnen
03.054 Versand für Auftrag 1 nach Wien begonnen. Dauer [Tage]: 2
03.644 Verpackung von Auftrag 3 durch B beendet, jetzt versenden
03.645 Auftrag 3 ins Ausland versendet
03.645 Verpackungsteam B ist fertig
04.365 Verpackung von Auftrag 4 durch C beendet, jetzt versenden
04.365 Auftrag 4 ins Inland versendet
04.366 Verpackungsteam C ist fertig
04.366 Versand für Auftrag 4 nach München begonnen. Dauer [Tage]: 1
05.065 Ware von Auftrag 1 in Wien ausgeliefert
05.066 Versand für Auftrag 3 nach Wien begonnen. Dauer [Tage]: 2
05.378 Ware von Auftrag 4 in München ausgeliefert
06.068 Verpackung von Auftrag 5 durch A beendet, jetzt versenden
06.068 Auftrag 5 ins Ausland versendet
06.069 Verpackungsteam A ist fertig
06.069 Logistiker (Inland) hat alle Versandaufträge abgeschlossen
07.067 Ware von Auftrag 3 in Wien ausgeliefert
07.068 Versand für Auftrag 5 nach Edinburgh of the Seven Seas begonnen. Dauer [Tage]: 30
37.076 Ware von Auftrag 5 in Edinburgh of the Seven Seas ausgeliefert
37.077 Logistiker (Ausland) hat alle Versandaufträge abgeschlossen
37.077 Alles erledigt
```

Simulation von Verpackung und Versand, hier mit verschiedenen Channels umgesetzt (Bild 2)

fen und über eine *await-foreach*-Schleife abgearbeitet. Gibt es nichts zu lesen, wird die Ausführung des laufenden Task-Objekts unterbrochen und, sobald wieder Daten in der Queue sind, fortgesetzt. Die Schleife wird automatisch verlassen, wenn über den Writer die *Complete*-Methode aufgerufen und damit der Channel geschlossen wurde und alle Elemente aus dem Channel verarbeitet worden sind. So lässt sich der Lesevorgang syntaktisch sehr kompakt und elegant umsetzen.

Alternativ kann man auch hier wieder selbst die Kontrolle übernehmen und mit Methoden wie *ReadAsync* oder *TryRead* einen einzelnen Eintrag entgegennehmen. Ein Aufruf von *TryPeek* erlaubt es, das nächste Element einzusehen, ohne es aus der Queue zu entfernen. *WaitToReadAsync* gibt ein *ValueTask*-Objekt zurück, das die Verfügbarkeit neuer Daten signalisiert. Verschiedene Eigenschaften wie *Count*, *CanCount*, *CanPeek* oder *Completion* liefern weitere Informationen über den Status der Queue.

Nicht so viel auf einmal

Muss die Menge der Einträge in der Queue begrenzt werden, verwendet man einen BoundedChannel. Beim Anlegen via *Channel.CreateBounded* lassen sich wieder Optimierungen und Verhalten des Channels beeinflussen. Für Optimierungen kommen wie zuvor die Optionen (hier vom Typ *BoundedChannelOptions*) *SingleWriter*, *SingleReader* und *AllowSynchronousContinuations* zum Einsatz. Natürlich muss man auch festlegen, wie viele Einträge denn maximal möglich sein sollen (siehe Variablen-Deklarationen in Listing 5 ganz oben).

Über einen Parameter vom Typ *BoundedChannelFullMode* kann man aber zusätzlich noch festlegen, was beim Schreiben in die Queue passieren soll, wenn die maximale Anzahl von Einträgen bereits erreicht ist. Der Enumerationswert *BoundedChannelFullMode.Wait* (die Voreinstellung) sorgt dafür, dass *Writer.WriteAsync* ein noch nicht erfülltes Task-Objekt zurückgibt beziehungsweise *TryWrite* den Wert *false*. Alternativ stehen die Enumerationswerte *DropNewest*, *DropOldest* sowie *DropWrite* zur Auswahl, die zwar ein kontinuierliches Schreiben zulassen, dafür dann aber Einträge fallen lassen. So werden dann entweder das neueste oder das älteste Element in der Liste oder aber das neu zu schreibende verworfen. Für diese Fälle lässt sich auch eine Rückrufmethode definieren, die immer dann ausgeführt wird, wenn ein Element verworfen wurde.

Die Schreibvorgänge in der Methode *Verpacken* in Listing 5 werden also blockieren, wenn die jeweilige Queue (*VersandInland* und *VersandAusland*) gefüllt ist (siehe Bild 2). Aus Gründen der Übersichtlichkeit der Ausgaben wurde im Beispiel die Zahl der Bestellungen auf fünf begrenzt. Sie können das Demoprogramm aber leicht anpassen und diese Anzahl erhöhen. Dann werden die Abläufe und insbesondere die ▶

Warte-Vorgänge besser sichtbar.

Das Lesen aus der Liste erfolgt wieder analog zur oben beschriebenen Vorgehensweise bei einem UnboundedChannel. Wie in Listing 6 zu sehen ist, kann auch hier auf ein IAsyncEnumerable zurückgegriffen werden, das man über *Reader.ReadAllAsync* erhält. Oder man benutzt die oben beschriebenen Mechanismen, um die Zugriffe selbst zu kontrollieren.

```
Zum Beenden Eingabetaste drücken
00.007 Bestellungen entgegennehmen
00.013 Verpackungsteam A hat Arbeit begonnen
00.015 Verpackungsteam B hat Arbeit begonnen
00.016 Verpackungsteam C hat Arbeit begonnen
00.017 Logistiker (Inland) bereit zum Versenden der Pakete
00.018 Logistiker (Ausland) bereit zum Versenden der Pakete
00.322 Bestellung 1 mit 7 Artikeln (Spiegel,Hemd,Bild,Fernseher,Spiegel,Spiegel,Geschirr) nach Hamburg
00.341 Verpackung von Auftrag 1 durch A begonnen
00.646 Bestellung 2 mit 5 Artikeln (Fernseher,Fernseher,Spiegel,Ball,Bild) nach Köln
00.646 Verpackung von Auftrag 2 durch B begonnen
00.959 Bestellung 3 mit 7 Artikeln (Hemd,Fernseher,Spiegel,Geschirr,Hemd,Fernseher,Spiegel) nach Stanley
00.960 Verpackung von Auftrag 3 durch C begonnen
01.272 Bestellung 4 mit 1 Artikeln (Geschirr) nach Brüssel
01.585 Bestellung 5 mit 3 Artikeln (Fernseher,Ball,Geschirr) nach Bremen
01.587 Alle Bestellungen entgegengenommen

Abbruch aller Vorgänge
02.266 Verpackung von Auftrag 2 durch B beendet, jetzt versenden
02.275 Verpackungsteam C hat Arbeit abgebrochen
02.367 Verpackungsteam A hat Arbeit abgebrochen
02.368 Verpackungsteam B hat Arbeit abgebrochen
02.416 Logistiker ({name}) hat Arbeit abgebrochen
02.416 Logistiker ({name}) hat Arbeit abgebrochen
02.416 Vorzeitiger Abbruch
```

Auch an eine Möglichkeit zum vorzeitigen Abbruch sollte man denken (Bild 3)

Ungeduldigen Anwendern entgegenkommen

In realen Anwendungen wird es immer wieder vorkommen, dass irgendein Mechanismus klemmt, eine Queue nicht geschlossen wurde und ein Reader endlos darauf wartet, dass er neue Daten lesen kann. Dann sollte es möglich sein, die Vorgänge gezielt abbrechen zu können. Wie in .NET üblich, lässt sich für derartige Abbrüche ein CancellationToken verwenden. Im Beispiel wird über eine Instanz von *CancellationTokenSource* ein solches bereitgestellt und in den jeweiligen Aufrufen verwendet beziehungsweise übergeben. Alle asynchronen Aufrufe wie beispielsweise *Writer.WriteAsync* oder *Reader.ReadAllAsync* nehmen ein CancellationToken entgegen und beenden einen eventuellen Wartevorgang, wenn *CancellationTokenSource.Cancel* aufgerufen wurde. Das wird in der Beispielanwendung ausgeführt, wenn der Benutzer die Eingabetaste betätigt. In Bild 3 sehen Sie eine Sequenz nach einem solchen Abbruch. Je nach Anwendungsart wird der Vorgang zum Abbrechen unterschiedlich zu implementieren sein. In einer WPF-Desktop-Anwendung könnte das zum Beispiel über ein Command ausgelöst werden, in einer Webanwendung über das CancellationToken der *Controller*-Methode.

Wichtig ist, dass Sie die CancellationTokens konsequent durchreichen, sodass alle Vorgänge, die man nicht selbst kontrolliert, bei Bedarf gestoppt werden. Auch wenn man das Generator-Pattern (siehe *yield return*) wie in Listing 2 gezeigt einsetzt, sollte man gegebenenfalls ein CancellationToken vorsehen. Bei *IAsyncEnumerable*-Methoden muss der betreffende Parameter zusätzlich mit dem *EnumeratorCancellation*-Attribut gekennzeichnet werden.

Listing 7: Demo-Bestellungen

```
// Das Hauptprogramm startet die verschiedenen Tasks
// zur Abarbeitung der Demo-Bestellungen
private static async Task BeispielMitChannels()
{
  AbbruchDurchBenutzerÜberwachen();

  var tasks = new List<Task>();
  var tasksVerpackungsteam = new List<Task>();

  tasks.Add(BestellungenVerarbeiten());

  tasksVerpackungsteam.Add(
    VerpackungsteamStarten("A"));
  tasksVerpackungsteam.Add(
    VerpackungsteamStarten("B"));
  tasksVerpackungsteam.Add(
    VerpackungsteamStarten("C"));

  tasks.Add(PaketversandAusführen(
    "Inland", channelVersandInland));
  tasks.Add(PaketversandAusführen(
    "Ausland", channelVersandAusland));

  await Task.WhenAll(tasksVerpackungsteam);
  channelVersandAusland.Writer.Complete();
  channelVersandInland.Writer.Complete();

  await Task.WhenAll(tasks);

  if (cancellationToken.IsCancellationRequested)
    Print("Vorzeitiger Abbruch", ConsoleColor.Red);
  else
    Print("Alles erledigt", ConsoleColor.Green);
}
```

Listing 8: Helferlein

```
// Ein paar Werkzeuge für die Konsolenausgabe des
// Demoprogramms.
// Zur Zeitmessung
static readonly Stopwatch stopwatch =
  Stopwatch.StartNew();
static readonly object syncObj = new();
...

static void Print(string text,
    ConsoleColor color=ConsoleColor.White)
{
  lock (syncObj)
  {
    var previousColor = Console.ForegroundColor;
    Console.ForegroundColor = color;
    Console.WriteLine($"
      {stopwatch.ElapsedMilliseconds:00,000}
      {text}");
    Console.ForegroundColor = previousColor;
  }
}

private static void AbbruchDurchBenutzerÜberwachen()
{
  Task.Run(() =>
  {
    Console.WriteLine(
      "Zum Beenden Eingabetaste drücken");
    Console.ReadLine();
    Console.WriteLine("Abbruch aller Vorgänge");
    cancellationTokenSource.Cancel();
  });
}
```

Das Beispielprojekt

Die beschriebenen Methoden werden in einer Konsolenanwendung vom Hauptprogramm aus gestartet (Listing 7). Sie geben jeweils Task-Objekte zurück. Im Fall des Verpackungsteams sind das drei Tasks, die parallel ihre Aufgaben erfüllen können. Sind alle fertig (*await Task.WhenAll*), werden die Channels für den Versand geschlossen. Die beiden Versand-Tasks (je einer für In- und Ausland) lesen dann noch die verbliebenen Elemente aus den Queues und sind danach ebenfalls fertig.

Ein paar Helferlein für die Demo zeigt Listing 8. Den gesamten Code für eigene Experimente finden Sie unter [5].

Ist das ausgereift? Und wie ist die Performance?

Das Team, das die Channels implementiert hat, hat sicherlich mehr Know-how und Entwicklungsarbeit investieren können, als man für eigene Umsetzungen zur Verfügung hätte. Insofern werden die Channels, korrekte Anwendung vorausgesetzt, wesentlich sicherer und performanter sein als selbstgestrickte Lösungen.

Der Autor von [2] hat auch einige Benchmark-Tests durchgeführt. Im Vergleich zu anderen Techniken, wie sie beispielsweise in *System.Threading.Tasks.Dataflow* implementiert sind (insbesondere *BufferBlock<T>*), schneiden die Channels wesentlich besser ab (um mehr als eine Größenordnung schneller).

Noch mehr Komfort erwünscht?

Dann schauen Sie mal bei Erweiterungen wie *Open.ChannelExtensions* [6] vorbei. Hiermit lassen sich die einzelnen Aktionen in einer Fluent-Syntax als Pipeline aneinanderreihen. Je nach Komplexität der Aufgabenstellung lässt sich damit der eigene Code nochmals vereinfachen.

Fazit

Immer dann, wenn man für die Entkopplung asynchroner Vorgänge auf First-in-First-out-Listen zurückgreifen will, lassen sich die neuen Channel-Klassen hervorragend einsetzen. Sie bringen vieles mit, was man sonst mühsam selbst implementieren müsste. Dazu gehören beispielsweise die Synchronisationsmechanismen, das Benachrichtigen der Empfänger über die Verfügbarkeit neuer Daten oder Abbruchmöglichkeiten via CancellationToken.

Der Umgang mit den Channels ist dank der verschiedenen Zugriffsmethoden sehr einfach, sicher und elegant. Mit wenigen, übersichtlichen Codezeilen lassen sich Daten in eine Queue schreiben und wieder aus ihr lesen und trotzdem kann die Sicherheit bei asynchronen Zugriffen gewährleistet werden. Insofern sind die Channels eine sehr gute Wahl, wenn sie zur Aufgabenstellung passen. Jeder Entwickler sollte sie kennen und bei Bedarf darauf zurückgreifen. ■

[1] Channel<T> bei Microsoft Learn, www.dotnetpro.de/SL2304Channels1

[2] Stephen Toub, An Introduction to System.Threading.Channels, www.dotnetpro.de/SL2304Channels2

[3] Video Working with Channels in .NET, www.dotnetpro.de/SL2304Channels3

[4] Video C# Channels Explained (System.Threading.Channels), www.dotnetpro.de/SL2304Channels4

[5] Quellcode der Beispiele auf GitHub, www.dotnetpro.de/SL2304Channels5

[6] Open.ChannelExtensions auf GitHub, www.dotnetpro.de/SL2304Channels6

Joachim Fuchs

Dr. Joachim Fuchs ist begeisterter Anhänger von Microsofts .NET-Philosophie. Er arbeitet als Softwarearchitekt, Berater und Dozent bei www.it-visions.de. Seine Schwerpunkte liegen derzeit bei XAML- und Web-UI-Technologien.

dnp@fuechse-online.de

dnpCode A2304Channels